Scheduling CSP

Aravind Sundaresan – sundrsn2 (3 Units)
Krish Masand – masand2 (3 Units)

1.0 Setting up the Environment

To make our search method more intuitive, we created Class, Semester, and Simulation objects.

Class objects store all characteristics of a class, such as fall price, spring price, credit hours, and an ArrayList of prerequisites.

Semester objects store an ArrayList of Classes, an integer value which determines if it is a fall or spring semester (fall if not divisible by 2, spring if divisible by 2), the total number of hours that each Class in the semester adds up to, and the total cost that the cost of each Class in the semester adds up to.

Simulation objects contain most of the methods used for searching for classes, as Simulation objects set up the entire environment and instantiate the Class and Semester objects when necessary. A Simulation object is what takes in one of the actual text files provided, parses the data and assigns it to the right areas. It assigns the first value to an integer that holds the number of courses, the second value to the minimum credit requirement for each semester, and the third value to the maximum credit requirement for each semester. It similarly goes through every other line and assigns proper values. When creating Class objects, the methods creates the Class objects based on each line in the input file that describes the Class, and then adds that Class to an ArrayList of all Classes. It then revisits each Class in the ArrayList when it reaches the lines that assign prerequisites to each Class. It reaches the second to last line, sets the first value to the number of courses that the student is interested in, and then adds each value in the line after that to an ArrayList that contains the number of each course that the student is interested in. Lastly, it takes in the last line, which is assigned to an integer called “budget,” which is actually never used because we did not have to keep track of the budget.

CSP Classification:

Variables: Classes, Semesters

Constraints: Credit Hours, Prerequisites

Domain: Simulations

1.1 Creating the Schedule

Our backtracking search primarily uses two methods: The Simulation object's "*add*" and the Semester object's "*add*." These two methods are utilized by the Simulation object's "*schedule"* method to create a working schedule.

Semester's *add* operates by taking in a Class object and returning the Class object that it added to the Semester's *classes* ArrayList(they may not be the same Class). It then checks to see if adding the Class would overflow the Semester object that's adding the class (making sure that adding the class would not lead to the Semester object exceeding the maximum allowed credit hours). After it has checked for this, it then checks if any of the Class's prerequisites have already been added to that Semester. If they have, the method returns null, and what happens next in this situation is handled by Simulation's *add* (to quickly summarize, it adds that particular Class to the next possible Semester that does not already have any of its prerequisites). The method then checks if any of the Class's prerequisites have not been taken yet by checking the Class's remaining prerequisites. If there are prerequisites, the method is called recursively on the next prerequisite (this is why the original Class that the method takes in may not be the Class that the method returns). If there are no prerequisites left to be taken, the class is added to the Semester.

Simulation's *add* uses Semester's *add* along with other tools to add a Class to the next available Semester. It takes in a Class object and returns nothing. *add* starts off by checking to see if the ArrayList of Semester objects is empty, and if it is, it adds an empty Semester object to the ArrayList. It then adds a Class to the next available Semester using Semester's *add*. If the Class that is returned is null, we know that one of the Class's prerequisites is already in that available Semester, so we attempt to add it to the next available Semester that does not have one of its prerequisites. If one does not exist, a new Semester object is created, and the Class is put in that Semester. After a Class was successfully added, we add that Class to an ArrayList of Classes that have been taken.

Using these two methods, the *schedule* method first adds all of the courses that the student is interested in. These are added in order as provided in the input text file. During this process, each prerequisite for all of the courses is also added, as part of Semester's recursive *add* function. After this, it is possible that some of the Semesters that have been created may not fulfill the minimum number of credit hours. To fulfill the minimum number of credit hours, we then add any class that has not been added using another implementation of Simulation's *add* that never creates new Semesters (it takes in the Class we want to add and the number of the Semester we want to add it to, instead of just a Class object). After that class or its prerequisite has been added (depending on what Semester's recursive *add* function returns), we check to see if the minimum number of credit hours has been met. If it hasn't, we continue to add Classes (skipping a Class that might potentially go past the maximum limit instead of just hitting

the minimum number of credit hours). If it has, we move on the next Semester until every Semester is fulfilled.

1.2 Results

First Scenario:
2347 3
2 9 8
2 3 5
3 1 2 4
638 907 802

Second Scenario:
6432 3
4 8 19 12 7
6 11 14 1 4 15 3
3 13 1 6
1549 3391 1492

Third Scenario:
11010 5
4 27 8 13 28
4 20 25 8 13
5 14 28 12 3 13
4 30 15 7 3
4 16 21 17 8
2152 718 2809 2690 2641

Fourth Scenario:
15959 7
4 2 38 14 26
4 34 14 2 12
5 28 31 17 3 19
4 9 19 38 12
6 17 13 2 38 12 14
4 18 13 38 12
5 3 13 17 38 12
2218 1734 2855 1408 3193 1766 2785

War Game

Aravind Sundaresan – sundrsn2 (3 Units)
Krish Masand – masand2 (3 Units)

2.0 Setting up the Environment

We developed our environment intuitively by creating both Board, Square, Player objects, where the Board class allows the user to create a board with whatever starting values they want, and with all Squares on the board set to unclaimed. The Square objects were made with two parameters, the points on the Square, and who the square belonged to. The board was created using a 2 dimensional array of type Square. The Player class specified what color the player was, and what agent the player would be using to win the game, Minimax or Alpha−Beta. A Player and a Square could either be of color green or blue, and two agents are required to start a game. The Board could be initialized in any of the 5 boards that were specified to test.

Boards are constructed to be 6 by 6, and default two 1 point in each square, and the color "n", for no color. Throughout the game,

Minimax and AlphaBeta agents were created as separate classes, with MinimaxObjects and AlphaBetaObjects to aid in packaging data to return from methods. The Board was passed into both of these agents, and they were able to make "moves" to modify the board on each of their turns. They made moves by executing the takeOver() method in the Board class, allowing them to take over a Square on the Board.

Simulations were executed by constructing 5 different boards: Keren, Narvik, Sevastopol, Smolensk, and Westerplatte. 4 pairs of agents for each board were created for a total of 40 simulations: Minimax vs. Minimax, Alpha-Beta vs Alpha-Beta, Minimax vs Alpha-Beta (Minimax first), and Minimax vs Alpha-Beta (Alpha-Beta first). These simulations recorded who won the game (obviously), the final score, the number of expanded nodes, the average nodes expanded per move, and the average amount of time per move in seconds.

2.1 Minimax and Alpha-Beta Agents

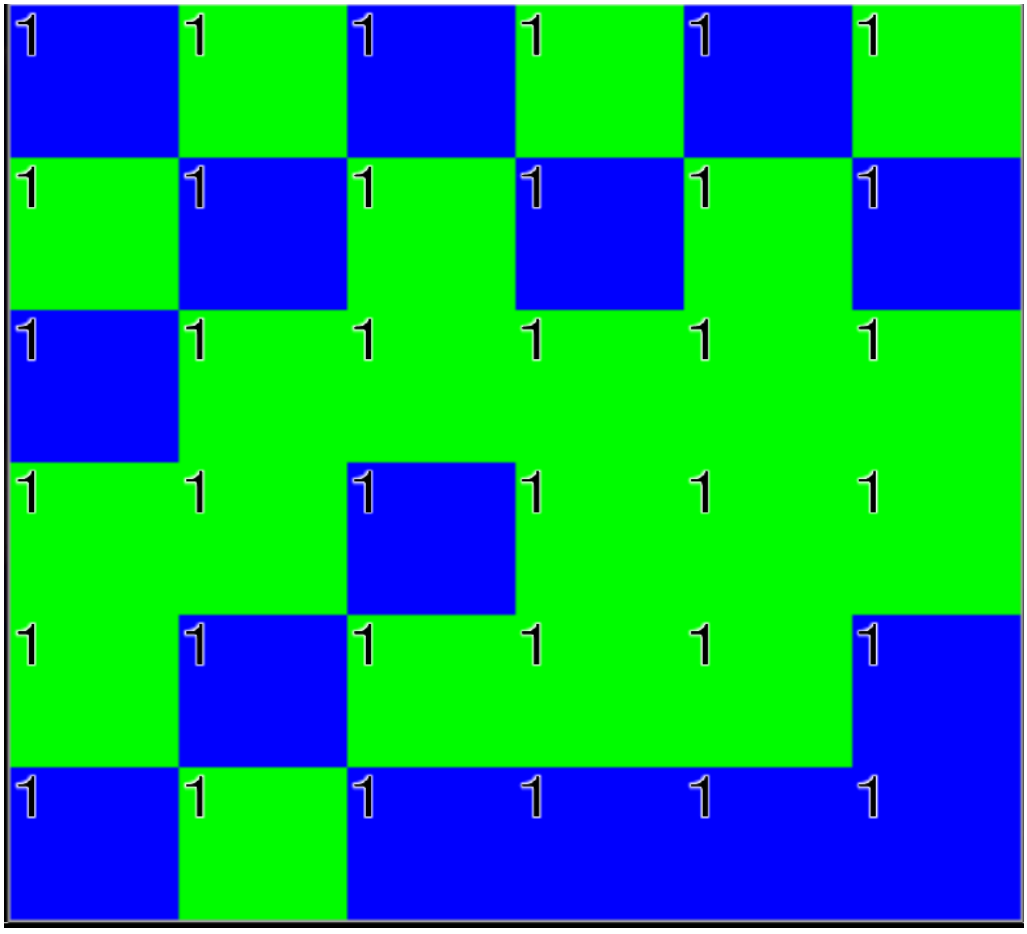
Here are the simulations for the 5 boards, and the results displayed numerically and graphically.

I. Keren

A) Minimax vs Minimax

Winner: Green

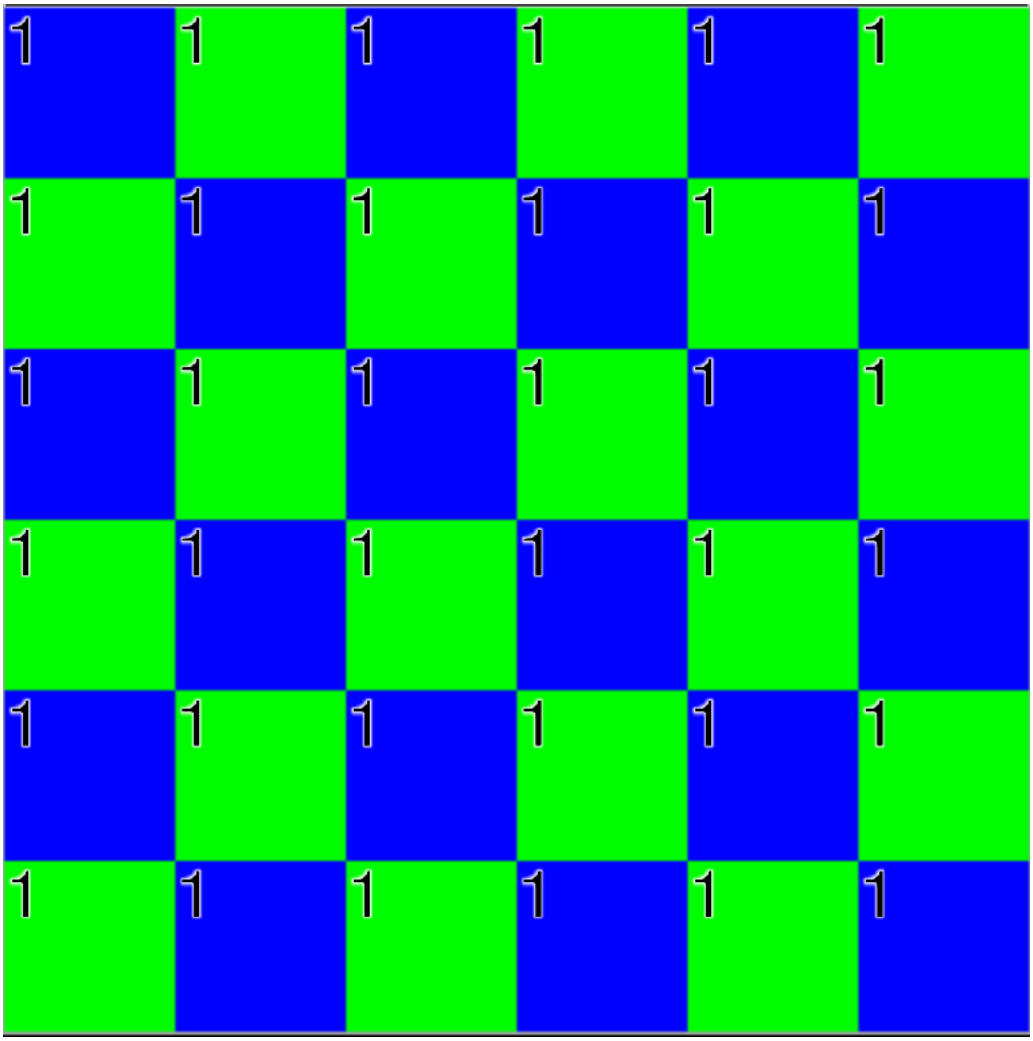
	Minimax (green)	Minimax (blue)
Final Score	21	15
Expanded Nodes	186967	209306
Average Nodes per Move	10387	11628
Average Time per Move	1.13	1.26



B) Alpha-Beta vs Alpha-Beta

Winner: Tie

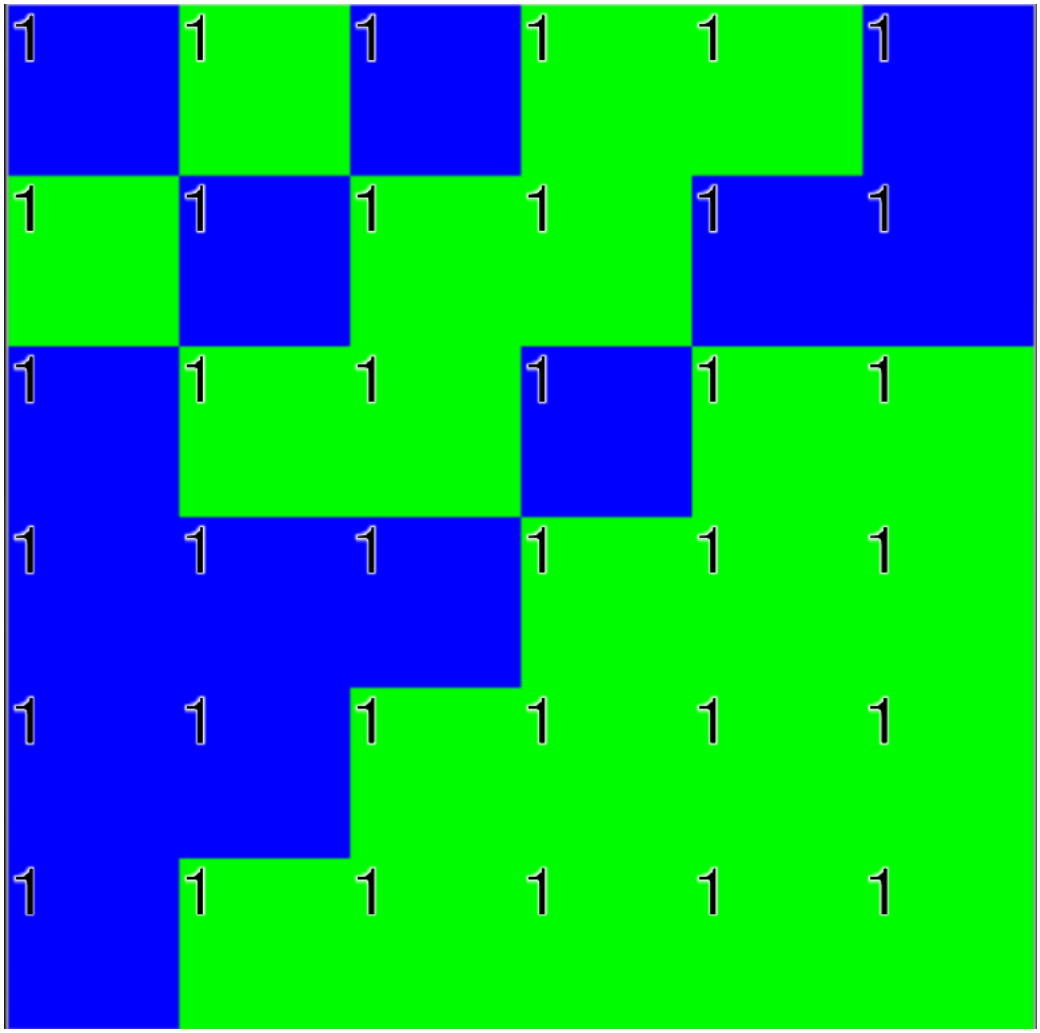
	Alpha-Beta (green)	Alpha-Beta (blue)
Final Score	18	18
Expanded Nodes	18391	22163
Average Nodes per Move	1022	1231
Average Time per Move	0.21	0.23



C) Minimax vs Alpha-Beta (Minimax first)

Winner: Green

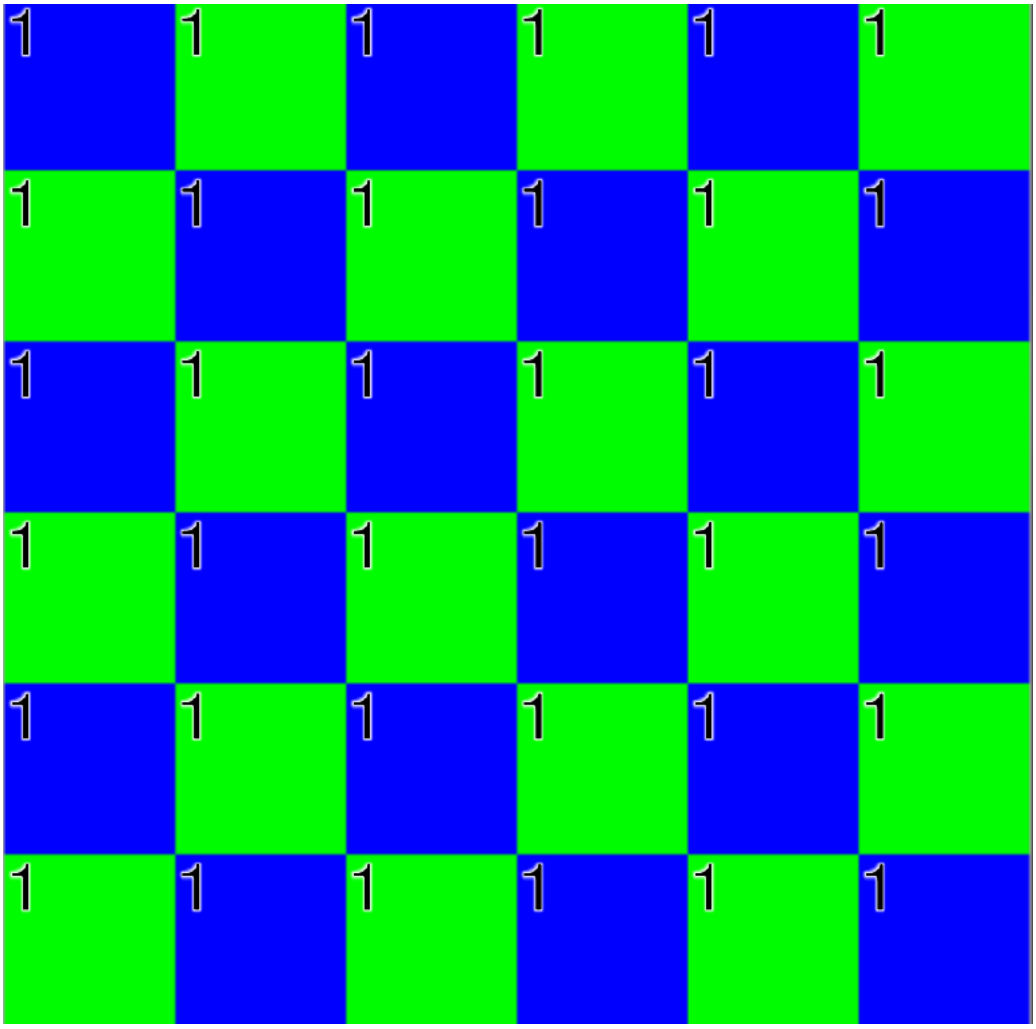
	Alpha-Beta (green)	Minimax (blue)
Final Score	22	14
Expanded Nodes	19956	209306
Average Nodes per Move	1109	11628
Average Time per Move	0.24	1.63



D) Alpha-Beta vs Minimax (Alpha-Beta first)

Winner: Tie

	Minimax (green)	Alpha-Beta (blue)
Final Score	18	18
Expanded Nodes	186967	22163
Average Nodes per Move	10387	1231
Average Time per Move	1.40	0.22

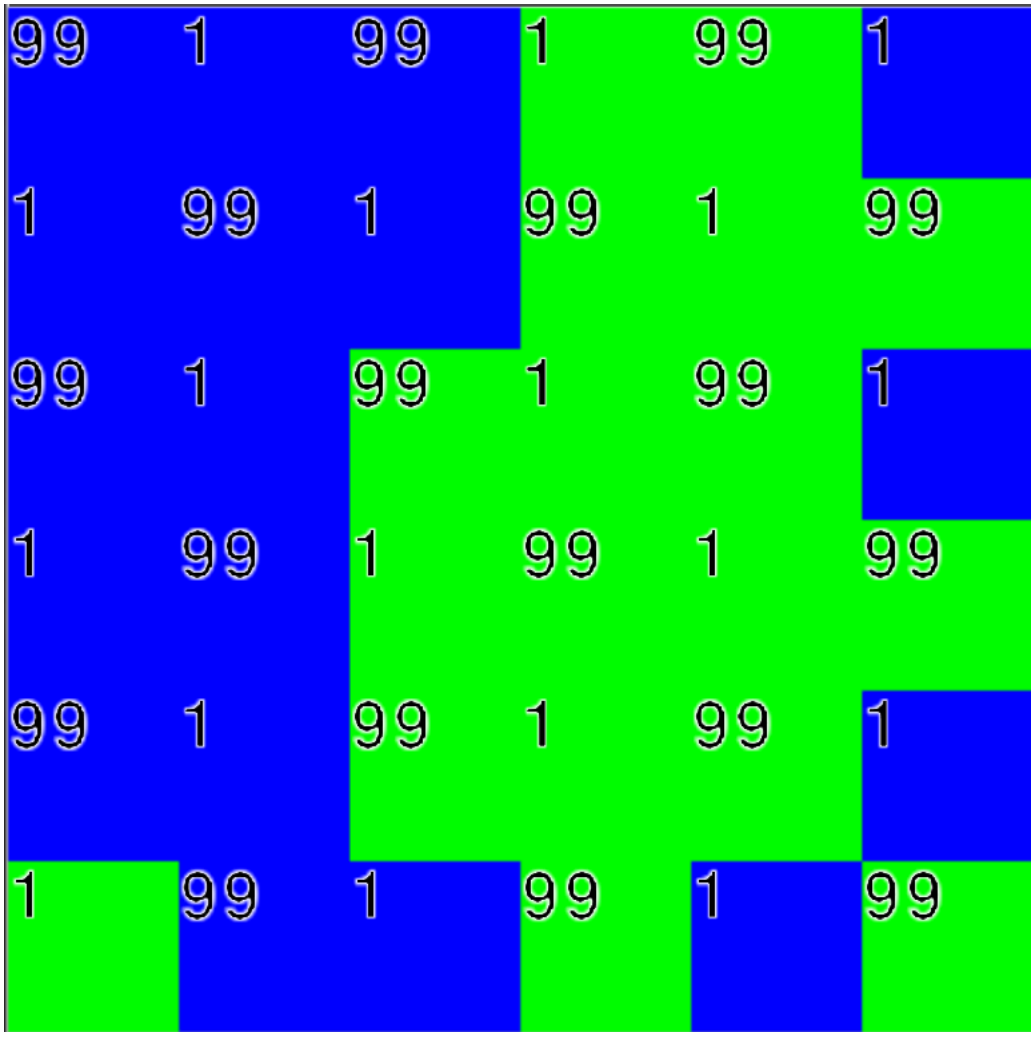


II. Narvik

A) Minimax vs Minimax

Winner: green

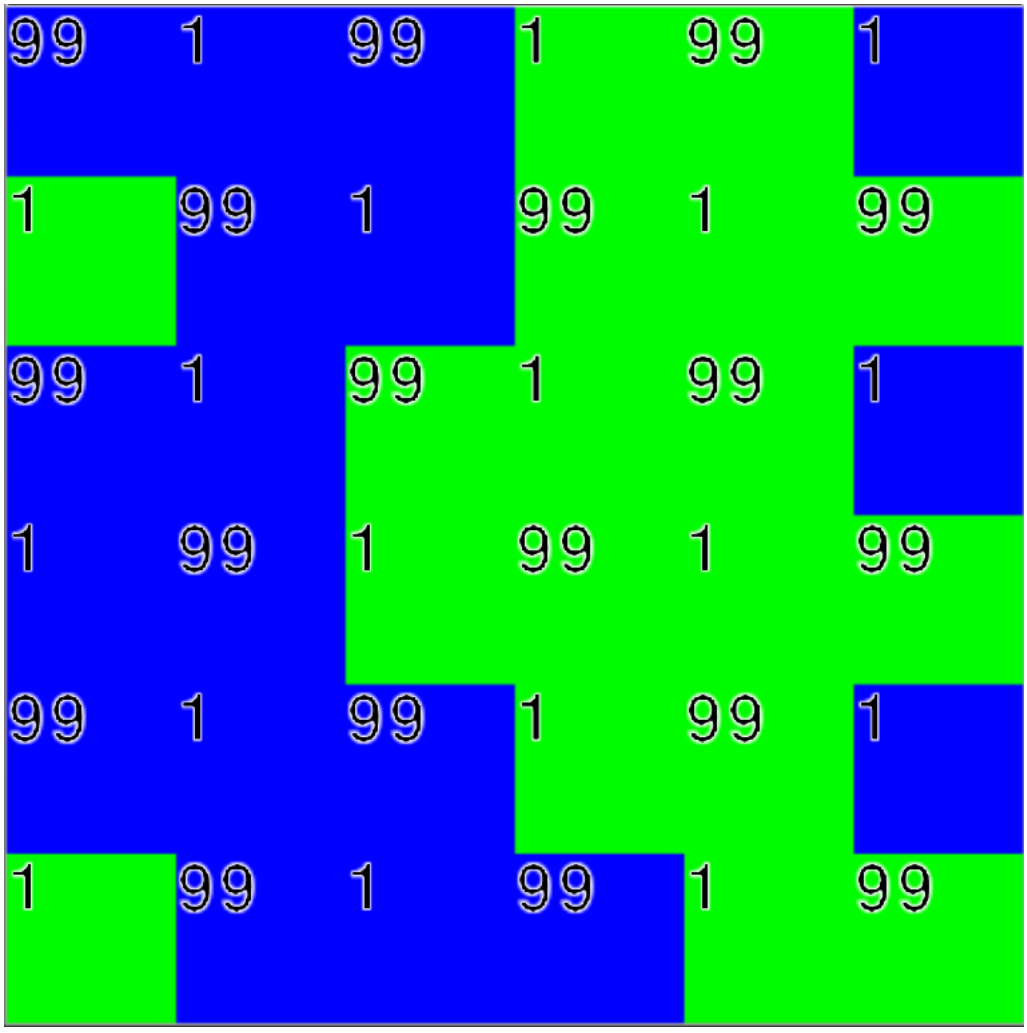
	Minimax (green)	Minimax (blue)
Final Score	1096	704
Expanded Nodes	186967	209306
Average Nodes per Move	10387	11628
Average Time per Move	1.16	1.23



B) Alpha-Beta vs Alpha-Beta

Winner: Tie

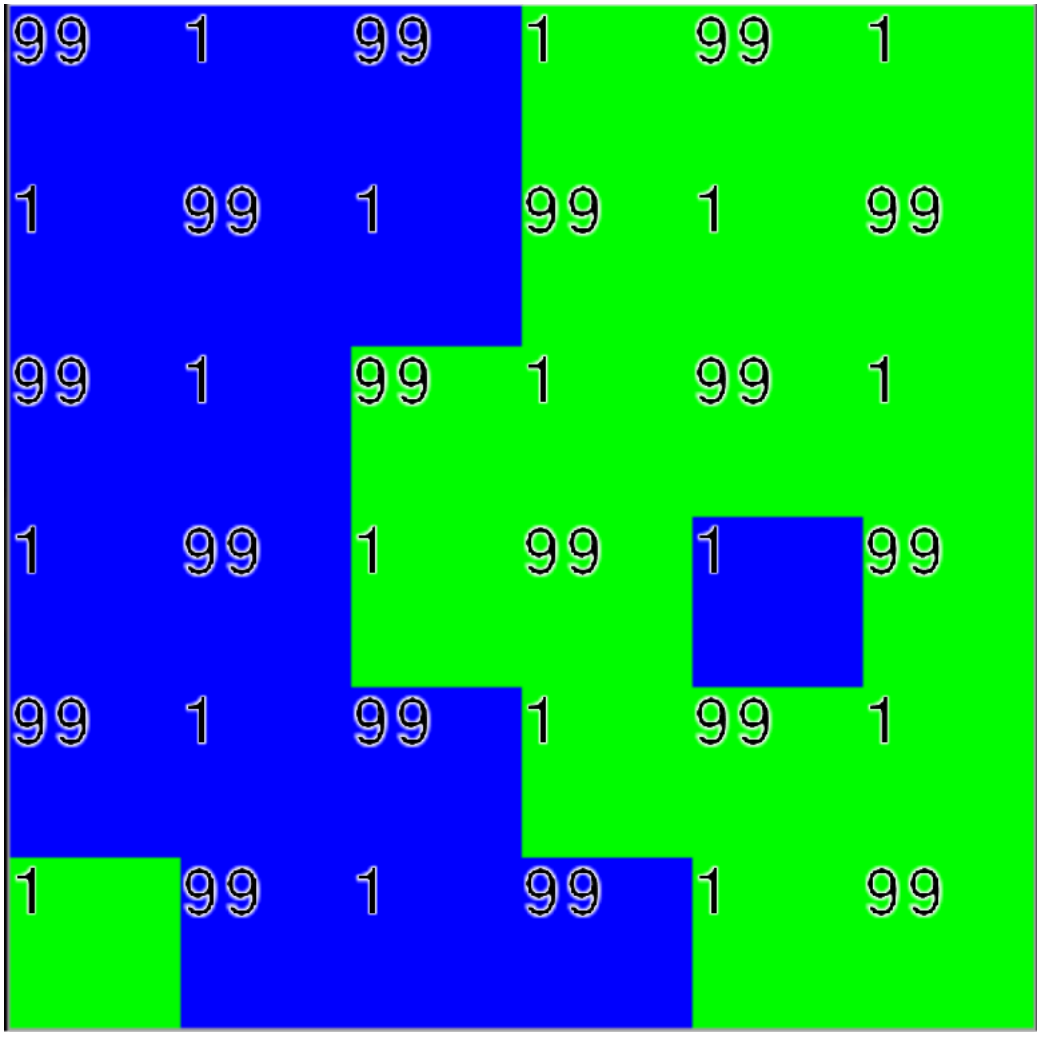
	Alpha-Beta (green)	Alpha-Beta (blue)
Final Score	900	900
Expanded Nodes	59725	52294
Average Nodes per Move	3318	2905
Average Time per Move	0.54	0.52



C) Minimax vs Alpha-Beta (Minimax first)

Winner: green

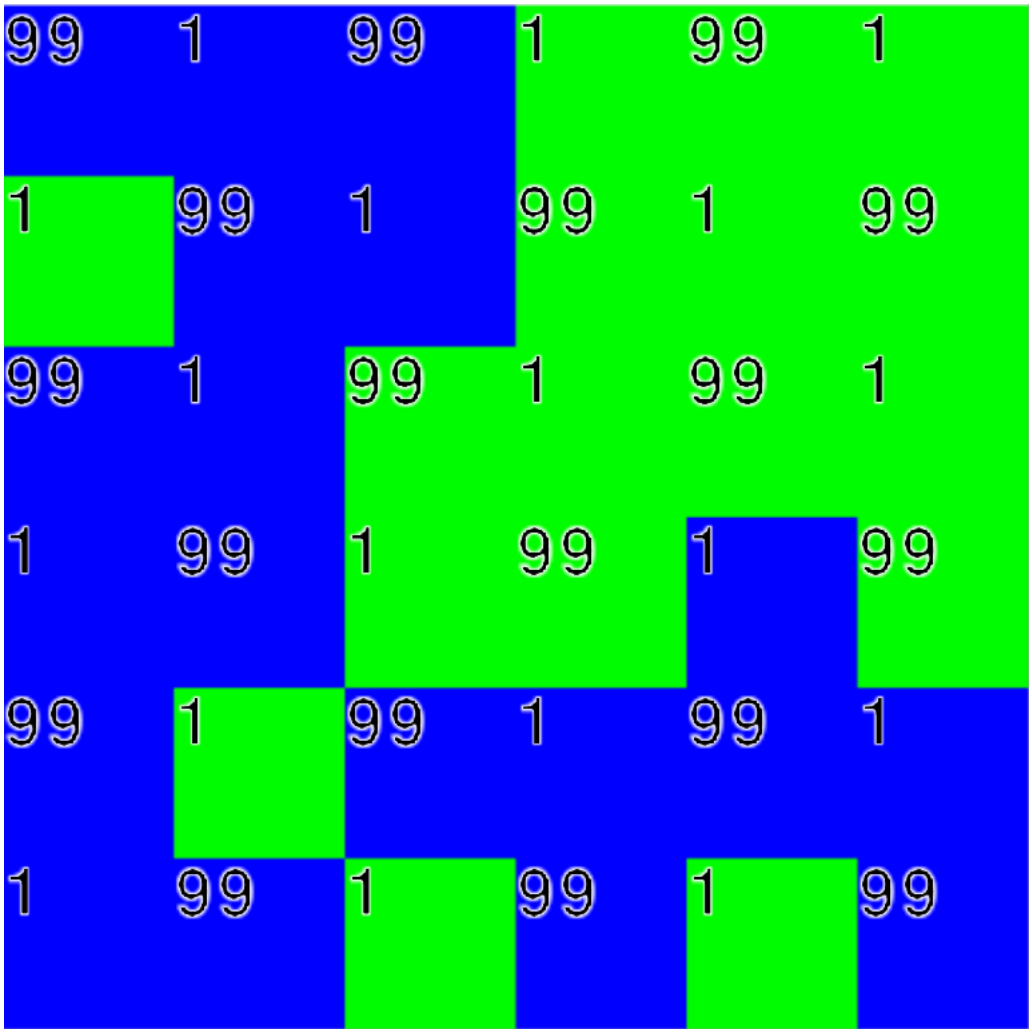
	Alpha-Beta (green)	Minimax (blue)
Final Score	901	899
Expanded Nodes	57782	209306
Average Nodes per Move	3210	11628
Average Time per Move	0.66	1.74



D) Alpha-Beta vs Minimax (Alpha-Beta first)

Winner: blue

	Minimax (green)	Alpha-Beta (blue)
Final Score	703	1097
Expanded Nodes	186967	51055
Average Nodes per Move	10387	2836
Average Time per Move	1.53	0.34

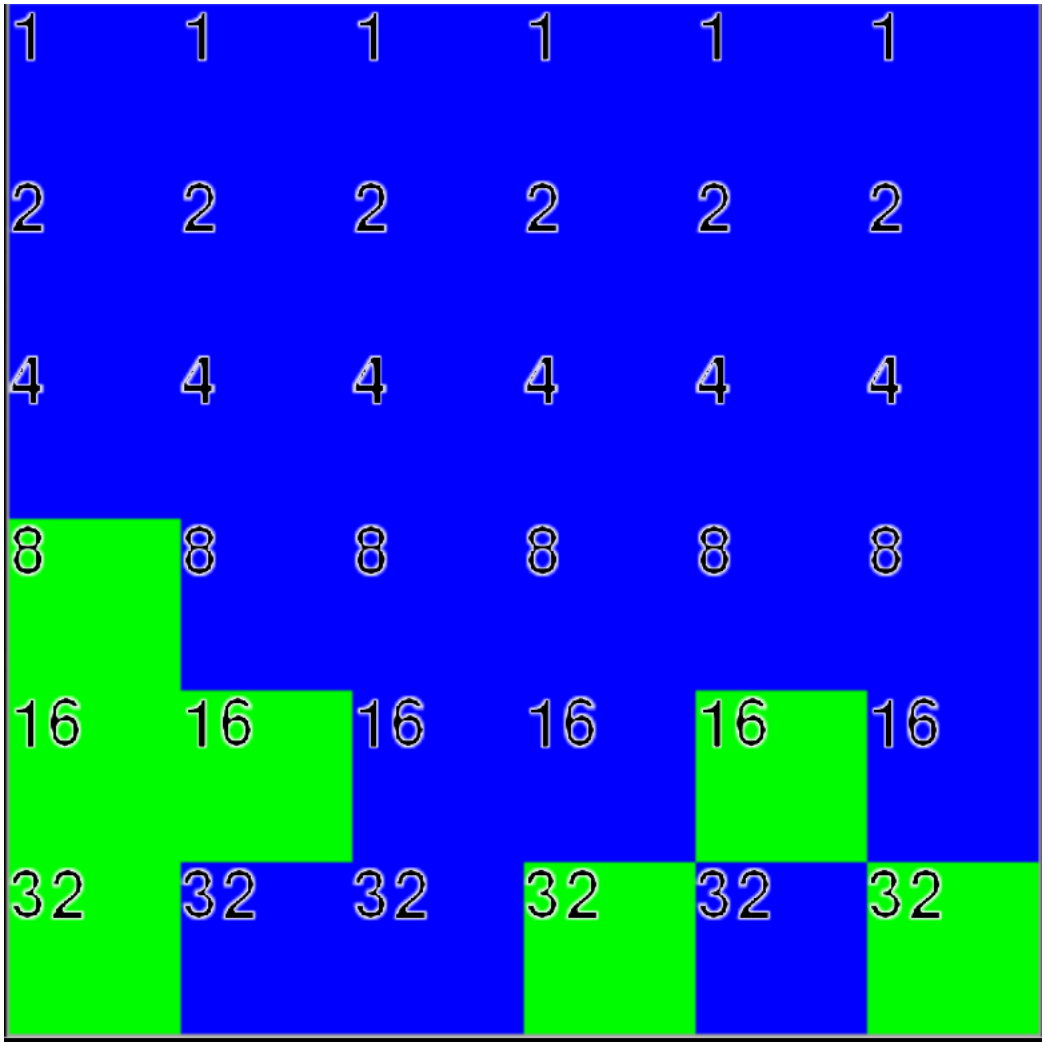


III. Sevastopol

A) Minimax vs Minimax

Winner: blue

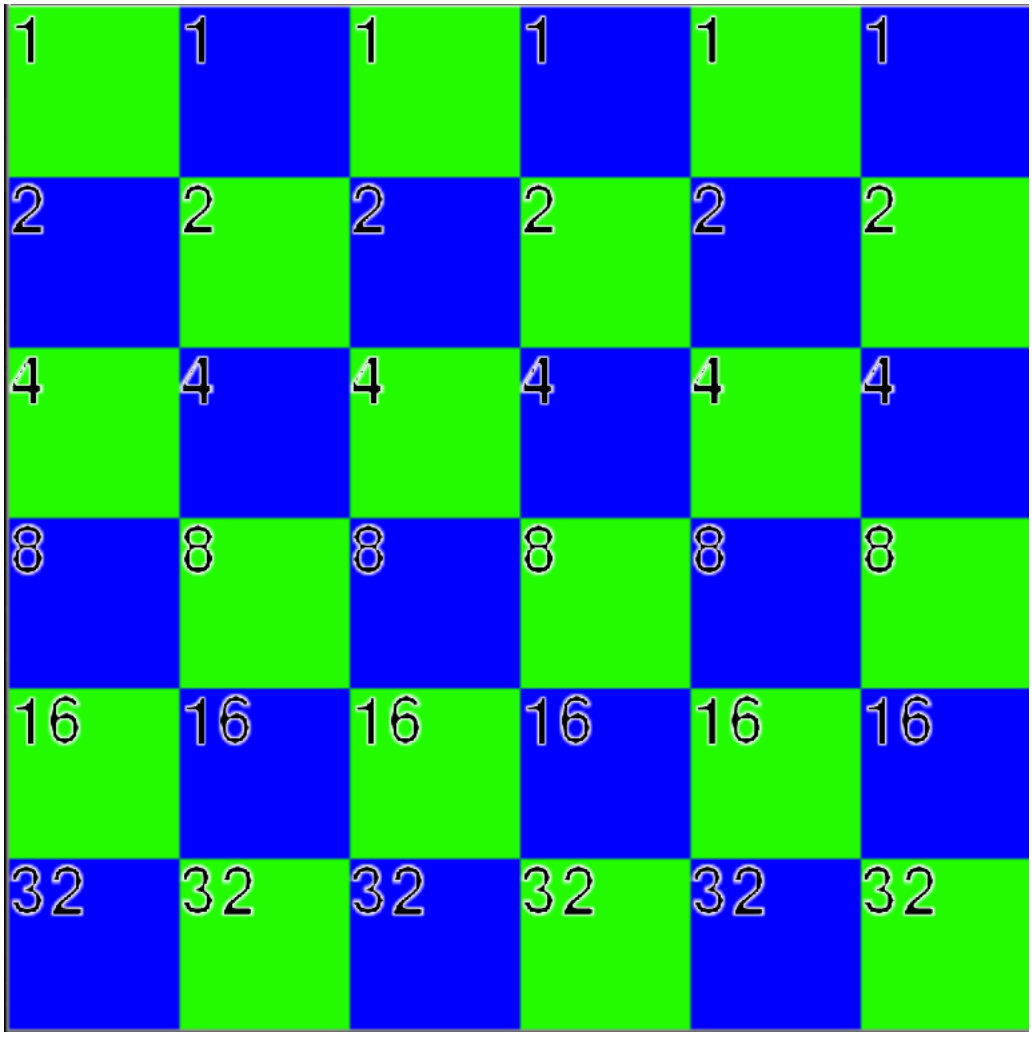
	Minimax (green)	Minimax (blue)
Final Score	152	226
Expanded Nodes	186967	209306
Average Nodes per Move	10387	11628
Average Time per Move	1.43	1.54



B) Alpha-Beta vs Alpha-Beta

Winner: blue

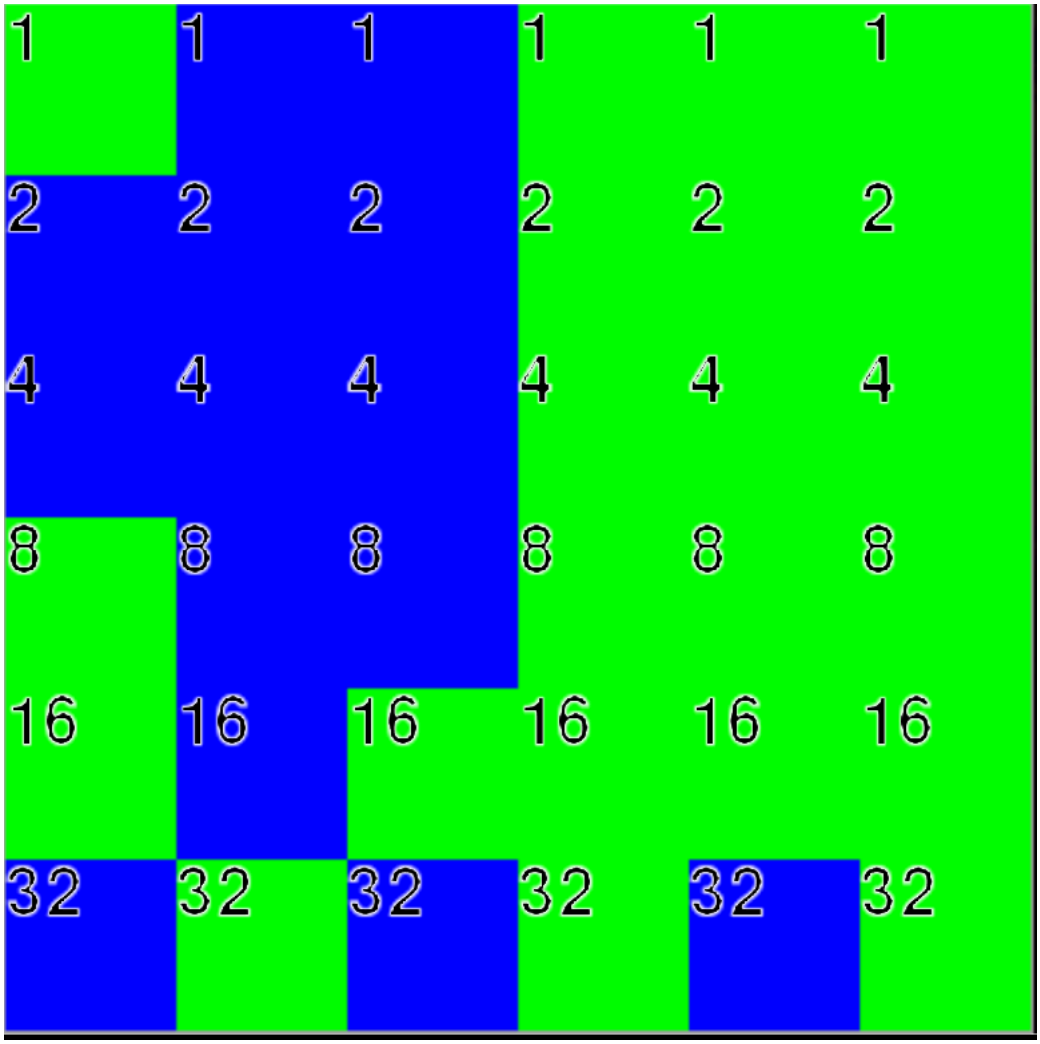
	Alpha-Beta (green)	Alpha-Beta (blue)
Final Score	188	189
Expanded Nodes	3256779	3610558
Average Nodes per Move	180932	200587
Average Time per Move	44.21	43.66



C) Minimax vs Alpha-Beta (Minimax first)

Winner: green

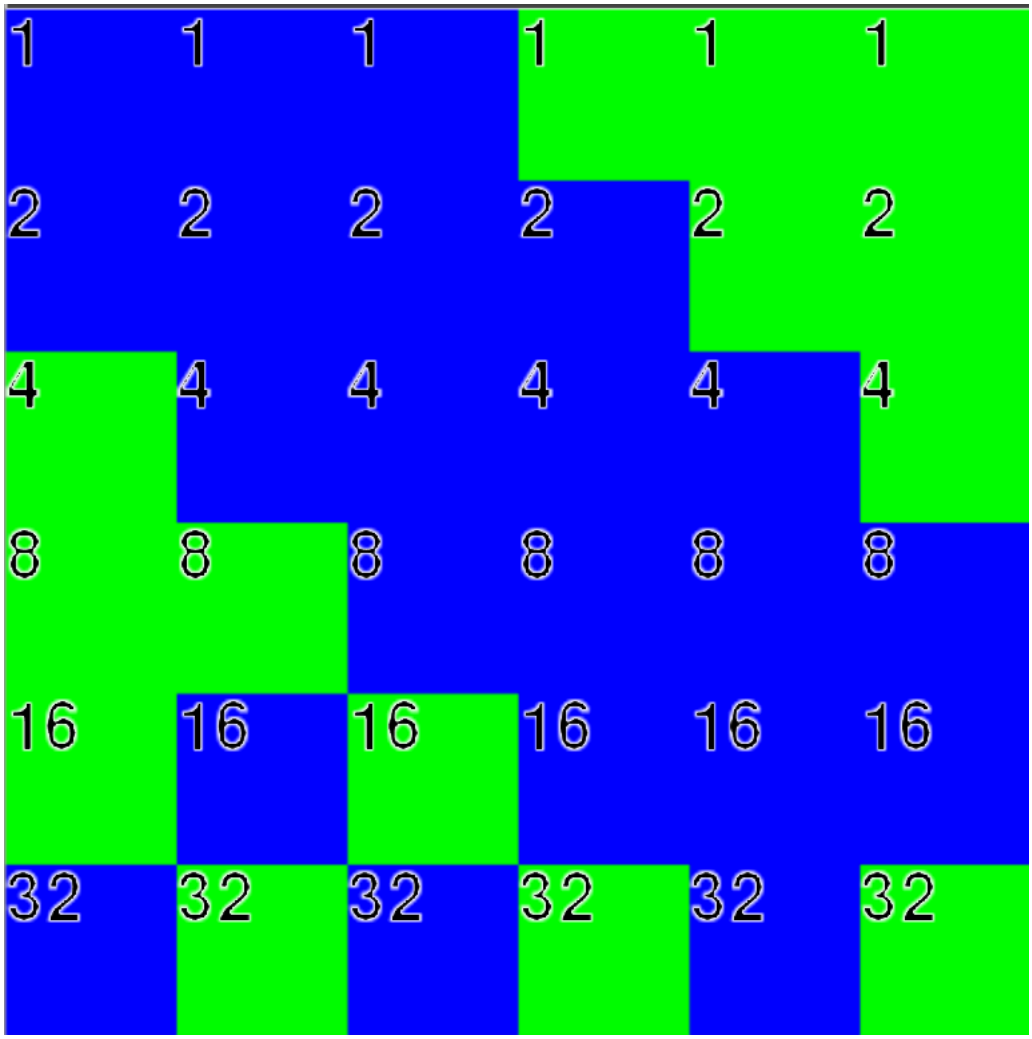
	Alpha-Beta (green)	Minimax (blue)
Final Score	230	148
Expanded Nodes	2917545	209306
Average Nodes per Move	162086	11628
Average Time per Move	44.60	3.16



D) Alpha-Beta vs Minimax (Alpha-Beta first)

Winner: blue

	Minimax (green)	Alpha-Beta (blue)
Final Score	159	219
Expanded Nodes	186967	3844125
Average Nodes per Move	10387	213562
Average Time per Move	1.16	30.17

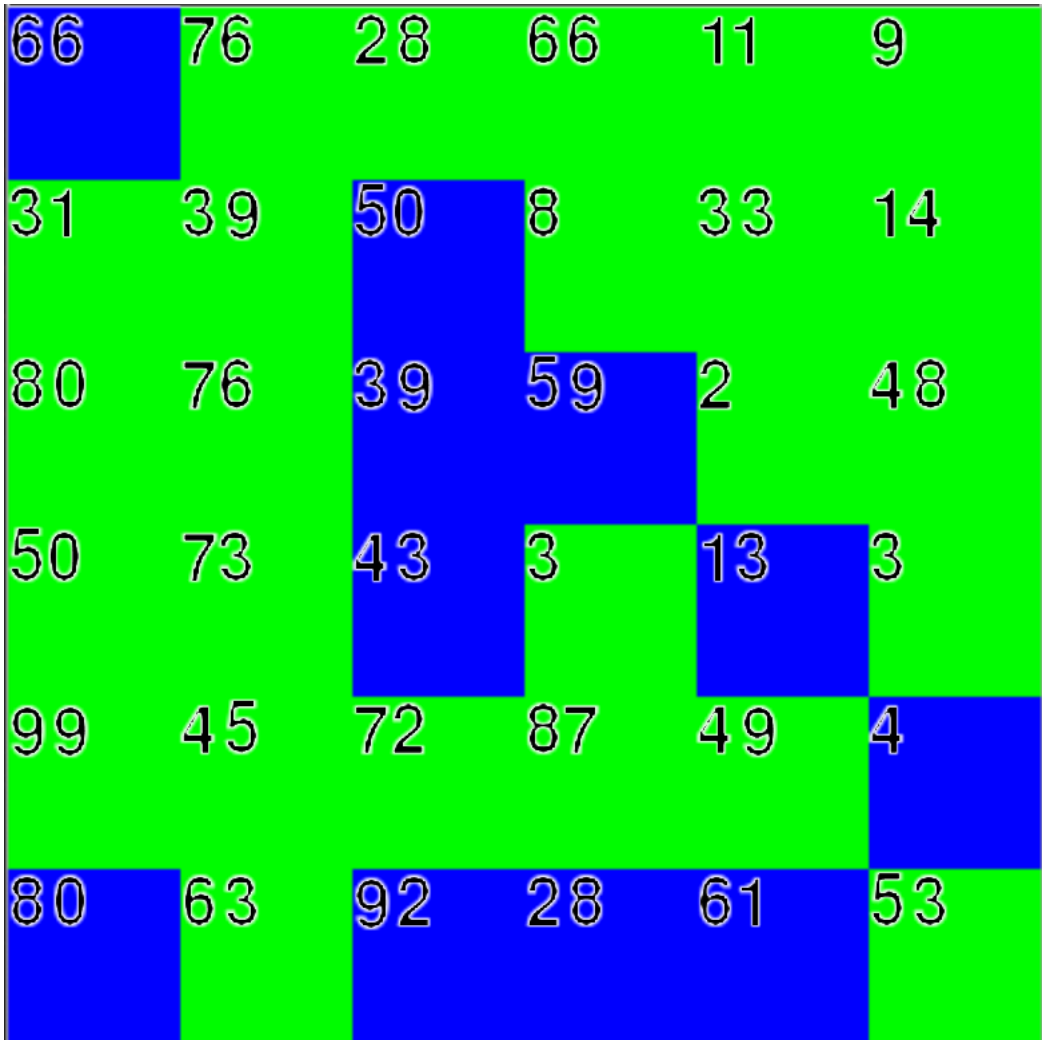


IV. Smolensk

A) Minimax vs. Minimax

Winner: green

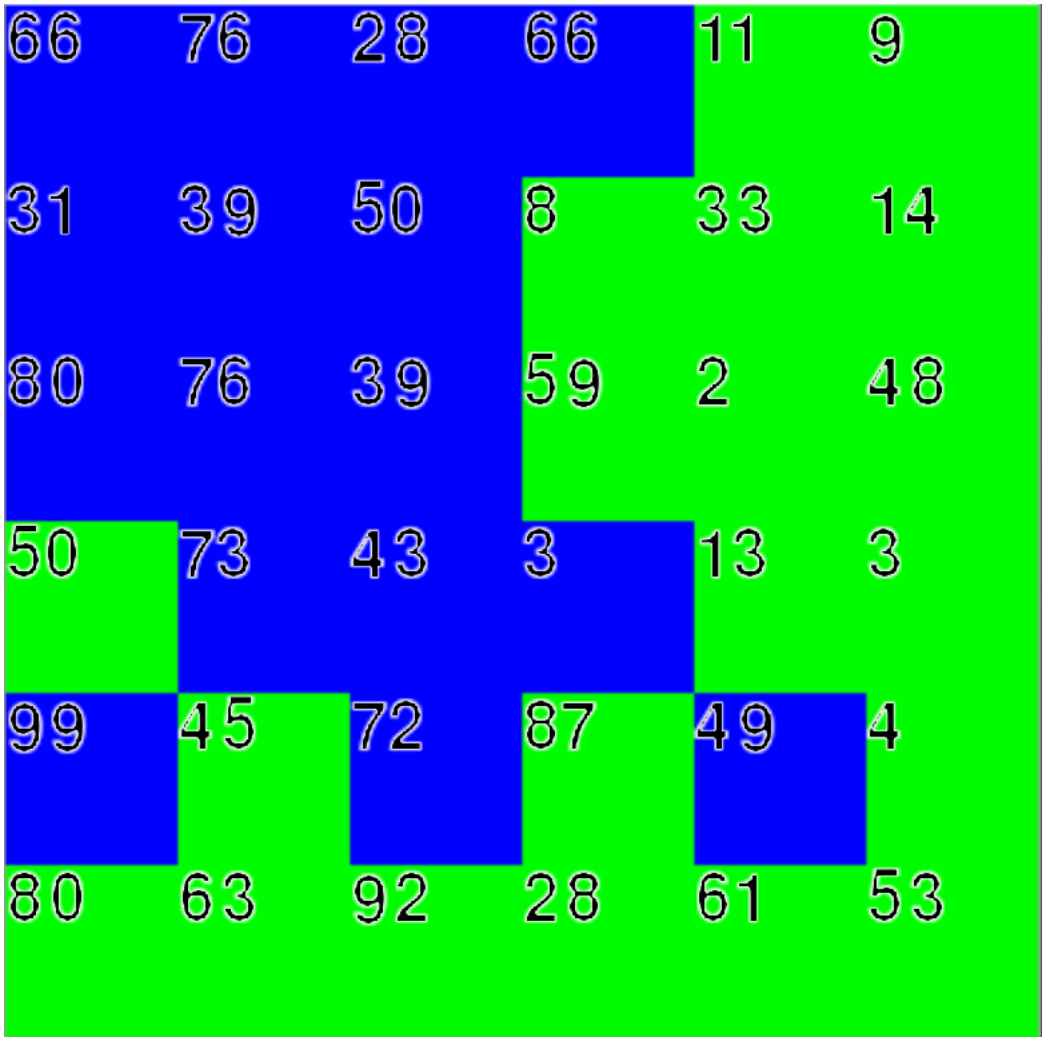
	Minimax (green)	Minimax (blue)
Final Score	1118	535
Expanded Nodes	186967	209306
Average Nodes per Move	10387	11628
Average Time per Move	1.14	1.30



B) Alpha-Beta vs Alpha-Beta

Winner: blue

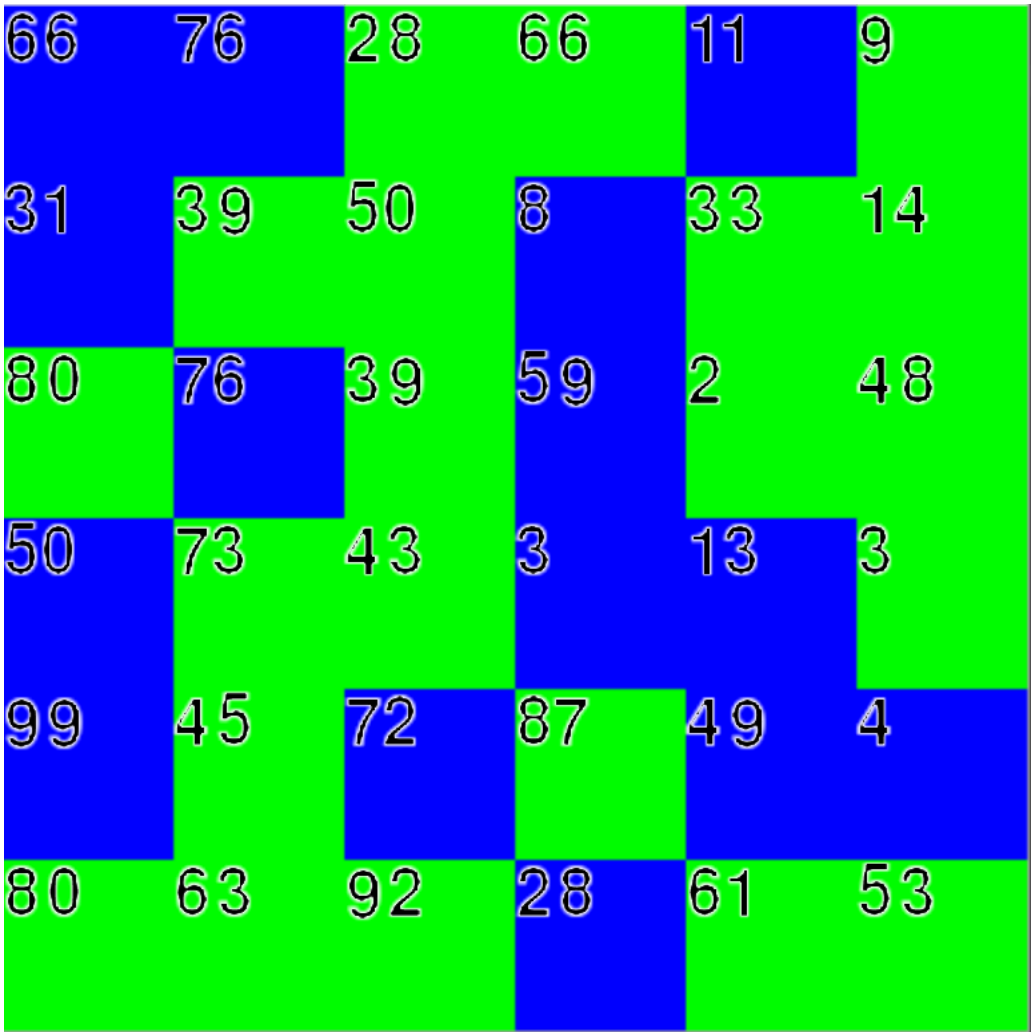
	Alpha-Beta (green)	Alpha-Beta (blue)
Final Score	763	890
Expanded Nodes	326671	236180
Average Nodes per Move	18148	13232
Average Time per Move	4.11	3.17



C) Minimax vs Alpha-Beta (Minimax first)

Winner: green

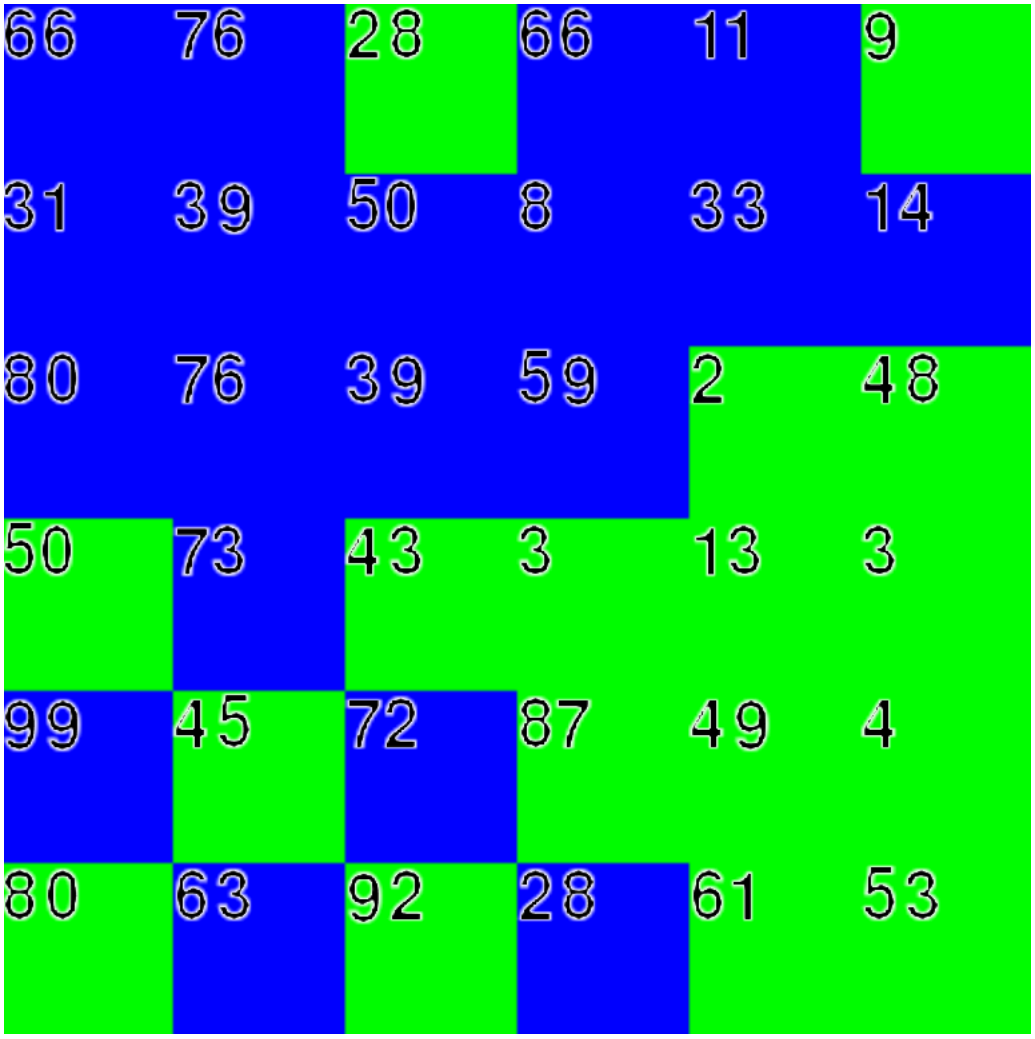
	Alpha-Beta (green)	Minimax (blue)
Final Score	1008	645
Expanded Nodes	760054	209306
Average Nodes per Move	42225	11628
Average Time per Move	7.89	2.23



D) Alpha-Beta vs Minimax (Alpha-Beta first)

Winner: blue

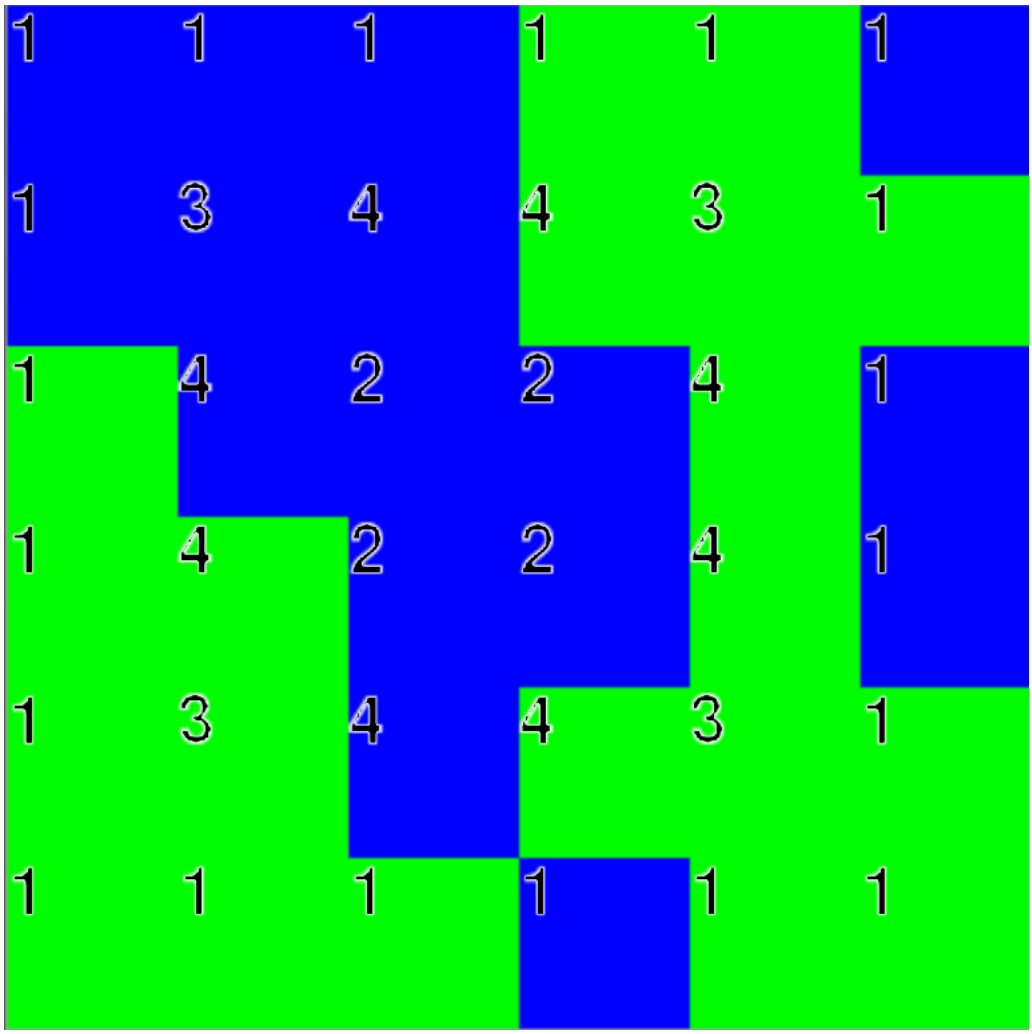
	Minimax (green)	Alpha-Beta (blue)
Final Score	670	983
Expanded Nodes	186967	196498
Average Nodes per Move	10387	10917
Average Time per Move	1.96	2.35



V. Westerplatte

A) Minimax vs Minimax
winner: green

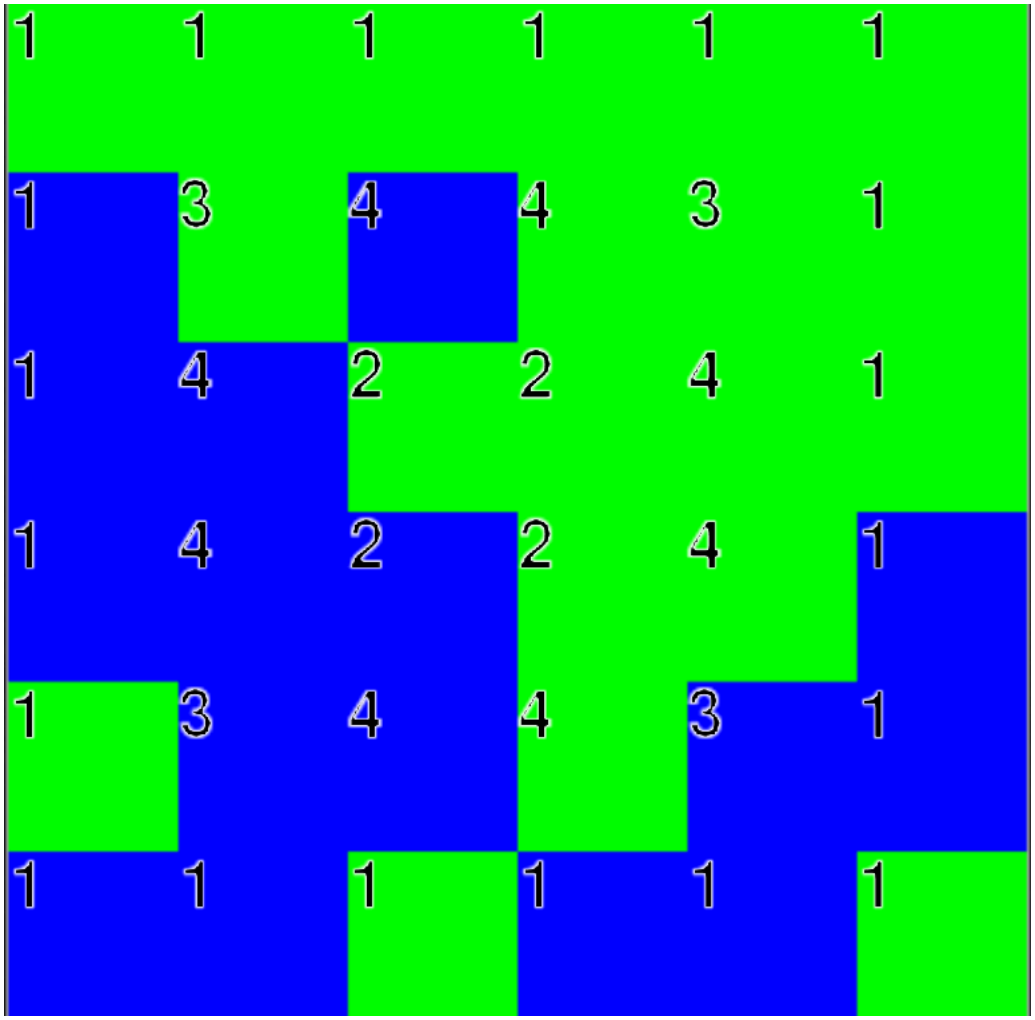
	Minimax (green)	Minimax (blue)
Final Score	41	31
Expanded Nodes	186967	209306
Average Nodes per Move	10387	11628
Average Time per Move	1.13	1.27



B) Alpha-Beta vs Alpha-Beta

Winner: green

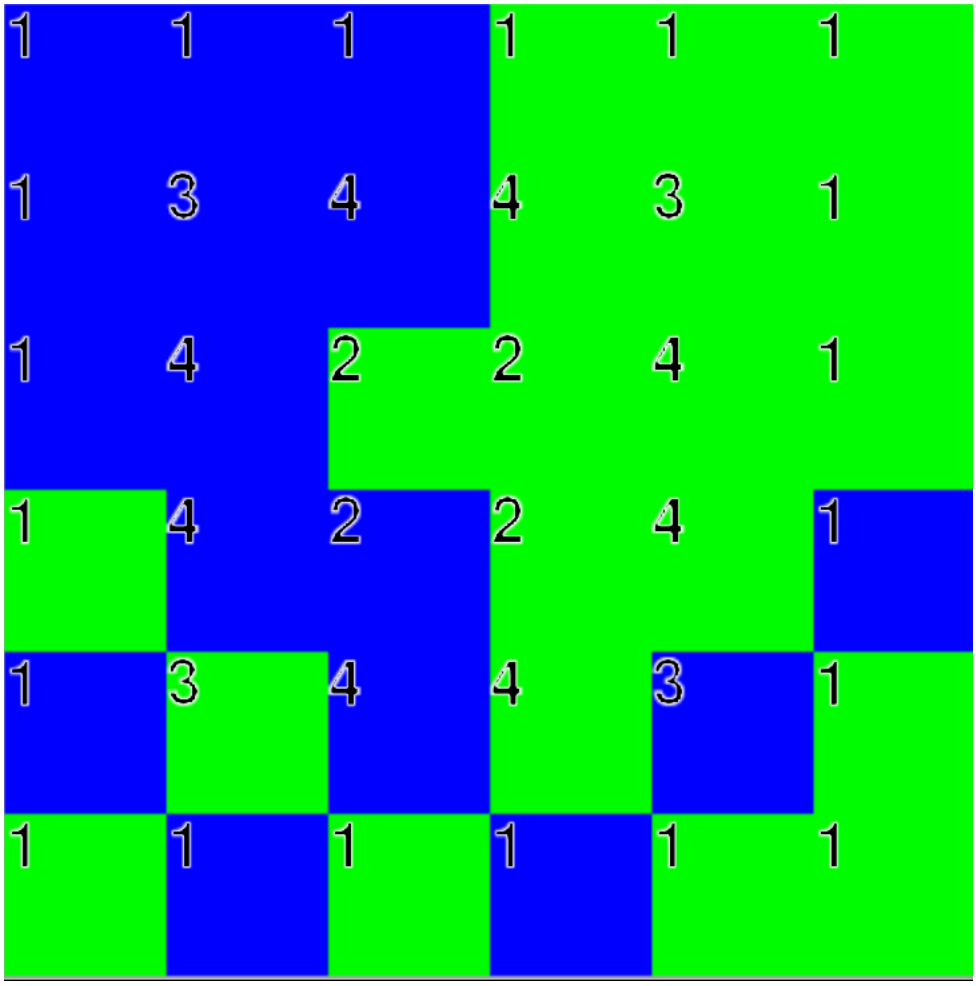
	Alpha-Beta (green)	Alpha-beta (blue)
Final Score	39	33
Expanded Nodes	151515	258006
Average Nodes per Move	8418	14334
Average Time per Move	1.67	2.54



C) Minimax vs Alpha-Beta (Minimax First)

Winner: green

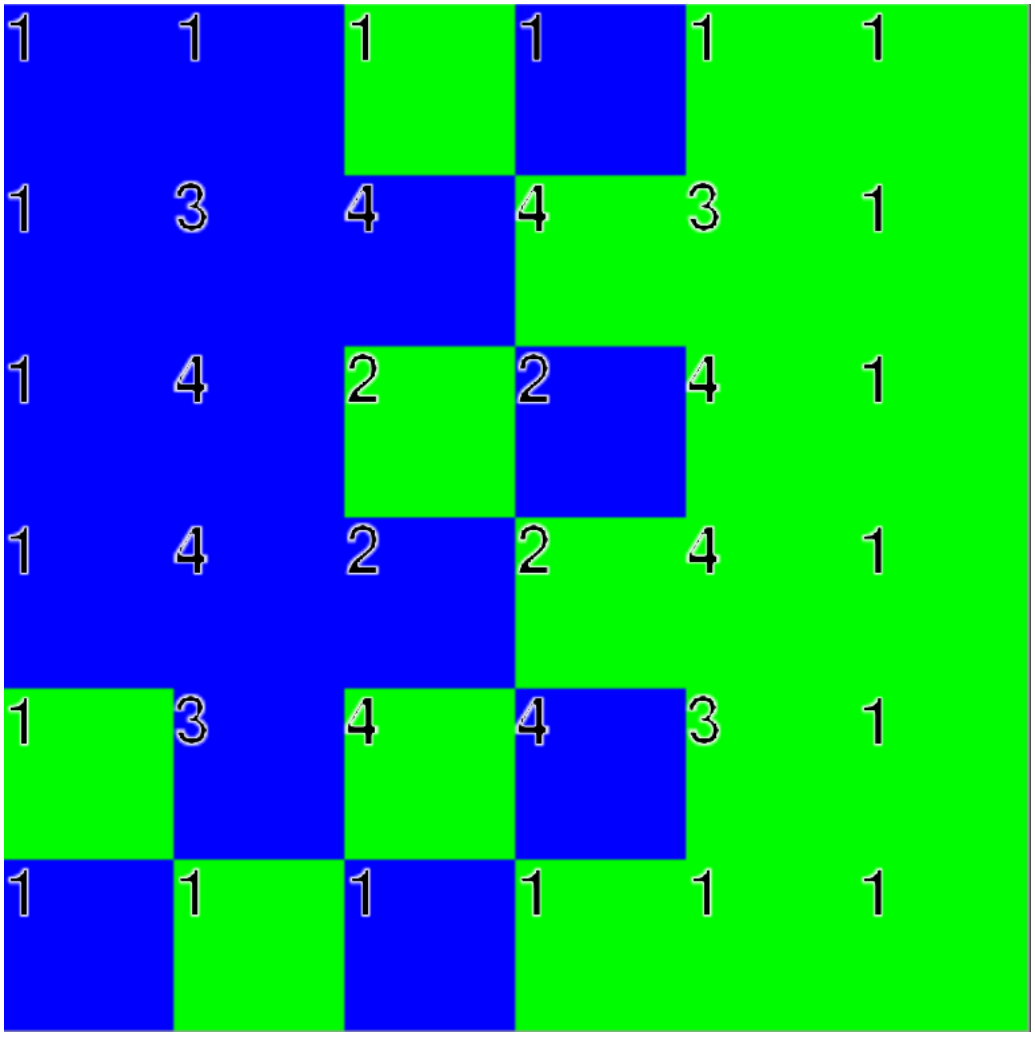
	Alpha-Beta (green)	Minimax (blue)
Final Score	39	33
Expanded Nodes	2248	209306
Average Nodes per Move	12493	11628
Average Time per Move	2.44	2.11



D) Alpha-Beta vs Minimax (Alpha-Beta first)

Winner: green

	Minimax (green)	Alpha-Beta (Blue)
Final Score	38	34
Expanded Nodes	186967	289998
Average Nodes per Move	10387	16111
Average Time per Move	2.05	3.21



On average, Alpha-Beta expanded much fewer nodes than Minimax did. The time taken for Alpha-Beta moves was also drastically lower than the move time for Minimax on higher-point boards. Alpha-Beta won more matchups than Minimax did, giving a safe assumption that Alpha-Beta pruning is a more efficient backtracking search algorithm than Minimax search.

Contributions

Krish Masand:

Krish developed all of part 1, proofed part 2 code, and wrote half of the report.

Aravind Sundaresan:

Aravind developed all of part 2, proofed part 1 code, and wrote half of the report.